1/3 T Software Equalizer Using Streaming SIMD Extensions

Version 1.1

01/99

Order Number: 243655-002

Table of Contents

Revision History

Revision	Revision History	Date
1.2	FCS version.	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. *Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture*

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provides 32-bit floating point single-instruction, multiple-data (SIMD) instructions. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, and spatial (3D) audio. This application note presents a method of implementing a 1/3 T software Equalizer, and includes examples of code that exploit the Streaming SIMD Extensions.

2 1/3 T Software Equalizer

Most data modems on today's market contain a hardware implementation of an equalizer. At the commencement of transmission, two modems establish a common ground on which the rest of the transmission is based. During this period of arbitration, the transmitting modem sends a prescribed data stream, which must then be interpreted by the receiving modem. Noise introduced into the signal during various stages of transmission often corrupts the data to an unrecognizable degree. To overcome "lossy" transmission, interpretation is done through a Least Mean-Squares (LMS) adaptive filter known as an "equalizer."

2.1 Applications for 1/3 T Software Equalizer

Knowing both the received data, *x(t)*, and the intended data, *y(t)*, the equalizer coefficients can be adapted over several iterations of the arbitration process, and coefficients characteristic of that particular transmission obtained. Coefficients can be affected by the transfer medium, modem implementation differences, and distance. Once obtained, equalizer taps are then constantly modified to maintain the established communication. The adaptation process can be a time-consuming one. An optimized implementation off-loaded to the processor could free modem resources normally used in synchronization.

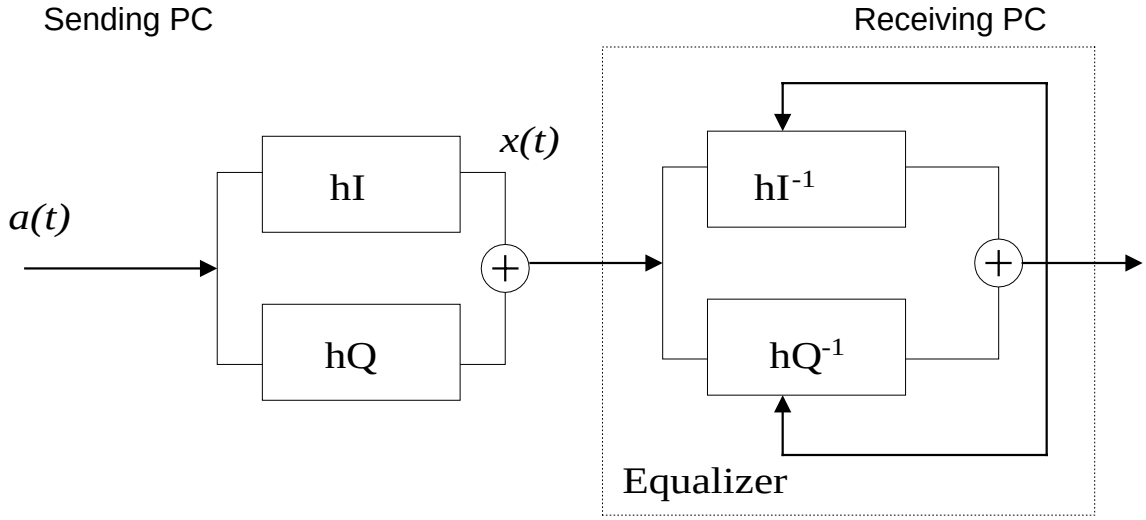

Figure 1: 1/3 T Equalizer block diagram

2.2 Implementing the 1/3 T Software Equalizer

The Streaming SIMD Extensions implementation of the 1/3 T Software Equalizer requires that input data be single-precision floating point values. Each input element must be separated into its real (Q) and imaginary (I) components and stored in a 16-byte aligned buffer as I, I, Q, -Q. These criteria are inherent in the filter algorithm. This implementation of the equalizer filter will fail if any of these is not met.

The filter coefficients are also single-precision floats divided into real and imaginary parts. Each "tap" is initialized to zero, except for the center-most tap, whose real and imaginary values begin at 0.5 for reference purposes. These coefficients are stored in a second memory buffer, in Q, I order, each Q-I pair being 8-byte aligned. Two Q-I pairs are then loaded simultaneously to obtain the preferred 16-byte alignment, while conserving memory storage requirements. The output values are once again separated in real and imaginary components. Some redundancy is inevitable with output values, which are stored in memory in I, Q, Q, I format. All constants are defined with the new "__m128" data type and stored in memory prior to use. They are then read in from 16-byte aligned locations as the algorithm dictates.

The use of the data structures is illustrated in Figure 2.

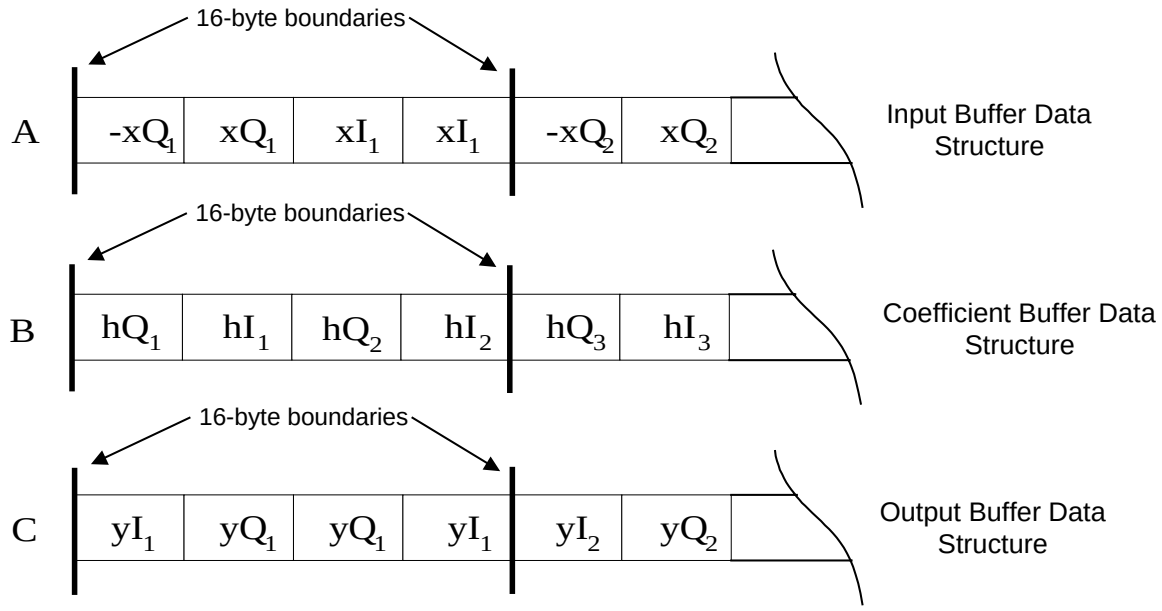

Figure 2: Buffer Data Structures

2.2.1 Techniques

The coefficients are 8-byte aligned rather than 16-byte aligned. Therefore, the coefficients must be loaded in and worked on in groups of two, to avoid misaligned access penalties. The code segment shown in Example 1 eliminates unaligned memory accesses by loading the coefficients *h1* and *h2* at their common 16-byte boundary, and avoids wasted bandwidth by using all memory elements as they are read.

```
movaps  xmm0, [esi][edx*2][n]     ; xmm0 <- | xI1 | xI1 | xQ1 |-xQ1 |
movaps  xmm1, [esi][edx*2][n+16]  ; xmm1 <- | xI2 | xI2 | xQ2 |-xQ2 |
movaps  xmm2, [eax][edx]          ; xmm2 <- | hI2 | hQ2 | hI1 | hQ1 |
movaps  xmm3, xmm2                ; xmm3 <- | hI2 | hQ2 | hI1 | hQ1 |
shufps  xmm2, xmm2, 0x44          ; xmm2 <- | hI1 | hQ1 | hI1 | hQ1 |
shufps  xmm3, xmm3, 0xEE          ; xmm3 <- | hI2 | hQ2 | hI2 | hQ2 |
mulps   xmm0, xmm2                ; xmm0 <- | xI1*hI1 | xI1*hQ1 | xQ1*hI1 |
                                            -xQ1*hQ1 |
mulps   xmm1, xmm3                ; xmm1 <- | xI2*hI2 | xI2*hQ2 | xQ2*hI2 |
                                            -xQ2*hQ2 |
addps   xmm7, xmm0
addps   xmm7, xmm1                ; xxm1 <- | sumI | sumQ | sumQ | sumI |
```

Example 1: Eliminating unaligned data accesses by loading multiple elements

For more information on data alignment, see the *Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, p. 5-1.

The equalizer modifies coefficients by transforming multiple input values and accumulating them into one output. Each output value's deviation from the expected output is calculated. The expected output values are limited to one of four quadrant center-points in the imaginary plane (±0.0625, ±0.0625). For example, if the input had a real value of 0.0105 and an imaginary value of 0.133, it would pertain to the first quadrant. The error (deviation) would be calculated by simply subtracting 0.0105 from +0.0625 (for the Q component) and subtracting 0.133 from 0.0625. Refer to Equation 1 and Figure 3.

$$y(i) = \Sigma\ x(3*i+n)h(n)$$

$$h'(n) = h(n) + k*[v(i)-y(I)]\ x(3*i+n)$$

Equation 1: Output and coefficient adaptation

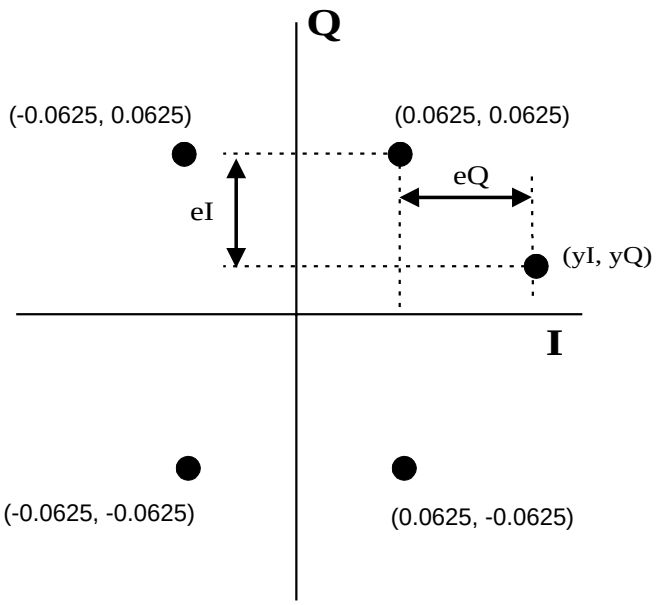

Figure 3: Calculating the error of a given output

Example 2 shows a segment of code that illustrates the generation of an error value based on previously calculated output values. Notice how the correct error can be determined without branching, simply by generating a Streaming SIMD Extensions mask and then "and"-ing and "or"-ing appropriately.

The assembly code in this example is not a direct continuation of the code in Example 1. For the complete code listing that shows the correct progression of instructions, refer to Section 6.

Original C implementation:

```
if (yI_float[some_index]>=0.0){vI=(float)0x0800/IFscale;}
else {vI=(float)-0x0800/IFscale;}
if (yQ_float[some_index]>=0.0){vQ=(float)0x0800/IFscale;}
else {vQ=(float)-0x0800/IFscale;}

eI=(vI-yI_float[j/3])/(2^16);                // step size
eQ=(vQ-yQ_float[j/3])/(2^16);                // step size
```

Corresponding Code Using Streaming SIMD Extensions:

```
xorps   xmm2, xmm2          ; zero out an xmm register for comparison to zero
movaps  xmm1, xmm0          ;copy the output result (yIQQI) from above
                            ; calculations
cmpps   xmm0, xmm2, 1       ; generate a "less than" mask
movaps  xmm5, xmm0          ; copy mask
movaps  xmm3, neg_const     ; load two alternative outcomes of
                            ; if..else statement
movaps  xmm4, pos_const
```

```
andps   xmm0, xmm3          ; vI and vQ calculated without conditional
                            ; branching
andnps  xmm5, xmm4
orps    xmm0, xmm5          ; xmm0 <- |  vI  |  vQ  |  vQ  |  vI  |

subps   xmm0, xmm1          ; xmm0 <- | vI-yI | vQ-yQ | vQ-yQ | vI-yI |
movaps  xmm5, step_size     ; xmm5 <- | step_size | step_size | step_size |
                            ;  step_size |
mulps   xmm0, xmm5          ; xmm0 <- | eI | eQ | eQ | eI |
```

Example 2: Calculating error values with C code vs. code using Streaming SIMD Extensions

This technique of mask generation can be useful for eliminating the penalty for a branch misprediction. Here it is ideal because there is no branch prediction pattern (the output could be anywhere in the imaginary plane) and the if.-else loops are small enough that calculating both results does not incur a greater penalty than misprediction. Further, masks and results for yI and yQ are calculated in the same xmm register, cutting the work in half. Also note that in the inline assembly version seen here, required bandwidth is reduced by not writing intermediate local variables (vI, vQ, eI, eQ) to memory in the middle of the loop.

Other conditional branching overhead is reduced by using only one loop counter as a data structure offset, and again eliminating unaligned memory accesses by operating on all loaded data elements at once. Some CPU cycles may be saved by implementing a backward branching loop (i.e. place the conditional jump at the bottom of the loop) rather than forward branching, as is implemented here. This would exploit the processor's inherent branch prediction algorithm. As it stands, the mere use of floating point over fixed point saves cycles in shifting and overflow issues.

Notice the shifting and conditional statements required in the following C code:

```
for (i=0; i<h_Leng; i++)
{ SumI = e.I*(long)sI[3+j+i]+e.Q*(long)sQ[3+j+i];
  SumQ = e.Q*(long)sI[3+j+i]-e.I*(long)sQ[3+j+i];
  SumI = ((SumI+0x4000)>>15)+hI[i];
  SumQ = ((SumQ+0x4000)>>15)+hQ[i];

  hI[i] = (short)SumI;
  hQ[i] = (short)SumQ;

  if (SumI>0x7fff)   { hI[i]=0x7fff; }
  if (SumI<-0x7fff)  { hI[i]=-0x7fff; }
  if (SumQ>0x7fff)   { hQ[i]=0x7fff; }
  if (SumQ<-0x7fff)  { hQ[i]=-0x7fff; }
}
```

as opposed to a few shuffle instructions in the Streaming SIMD Extensions assembly code:

```
I_LOOP:
   cmp   edx, last_h        ; check loop counter and jump conditionally
```

```
    jg    I_DONE
    movaps   xmm2, [esi][edx*2][48]   ; xmm2 <- | sI1 | sI1 | sQ1 | -sQ1 |
    movaps   xmm3, [esi][edx*2][64]   ; xmm3 <- | sI2 | sI2 | sQ2 | -sQ2 |
    movaps   xmm4, [eax][edx]         ; xmm4 <- | hI2 | hQ2 | hI1 | hQ1 |

    mulps    xmm2, xmm0      ; xmm2 <- | eI*sI1 | eQ*sI1 | eQ*sQ1 |-eI*sQ1 |
    mulps    xmm3, xmm0      ; xmm3 <- | eI*sI2 | eQ*sI2 | eQ*sQ2 |-eI*sQ2 |
    movaps   xmm5, xmm2      ; xmm5 <- | eI*sI1 | eQ*sI1 | eQ*sQ1 |-eI*sQ1 |
    shufps   xmm5, xmm5, 0x4E; xmm5 <- | eQ*sQ1 |-eI*sQ1 | eI*sI1 | eQ*sI1 |
    shufps   xmm2, xmm3, 0x44; xmm5 <- | eQ*sQ2 |-eI*sQ2 | eQ*sQ1 |-eI*sQ1 |
    shufps   xmm5, xmm3, 0xE4; xmm5 <- | eI*sI2 | eQ*sI2 | eI*sI1 | eQ*sI1 |

    addps    xmm5, xmm2
    addps    xmm4, xmm5        ; xmm4 <- | hI2' | hQ2' | hI1' | hQ1' |

    movaps   [eax][edx], xmm4  ; hIQ  <- | hI2' | hQ2' | hI1' | hQ1' |
    add      edx, 16
    jmp      I_LOOP
I_DONE:
```

Example 3: Adapting coefficients using C code and Streaming SIMD Extensions assembly code

3 Performance

3.2 Considerations

Originally, consideration was given to implementing the inline assembly version of the code with no modification of the algorithm whatsoever. This idea was quickly abandoned when the alignment issues were discovered. Though it could be implemented as such, it would require additional CPU cycles and memory bandwidth to move data into a temporary buffer to align data on a 16-byte boundary.

It was also observed that the algorithm benefits from loop unrolling to some degree. Both of the inner loops had heavy inter-loop dependencies, often relying on the result of a recently executed instruction for the next iteration to execute. Reducing the overhead of looping was a logical way to get more work done per loop, hence amortize the cost of looping over a broader range of instructions, and interleave the instructions to alleviate data dependencies.

As an aside, prefetching data elements was not examined for this kernel. It may be an interesting test case to experiment with cache hints, though the code appears to be more CPU-bound than memory-bound.

4 Conclusion

The Streaming SIMD Extensions implementation of the 1/3 T Equalizer algorithm reduces data accesses and avoids register “spills” by discarding intermediate values without storing. Loop unrolling and branch elimination (via mask generation) also improve overall code speed.

5 C Code Examples

Two versions of the 1/3 T Software Equalizer are shown in this section. Example 4 shows the entire algorithm in 32-bit fixed point C code. Example 5 shows the same algorithm implemented in 32-bit floating point C code. In both cases, the first inner loop is used to generate an output, which is then used by the outer loop to create an error signal to be used in updating the coefficients. The coefficients are then recalculated in the second inner loop using this error value.

```
for (j=0; j<x_Leng; j=j+3)
{  for (SumI=0, SumQ=0, i=0; i<h_Leng; i++)
   {
      SumI=SumI+sI[3+j+i]*(long)hI[i]-sQ[3+j+i]*(long)hQ[i];
      SumQ=SumQ+sI[3+j+i]*(long)hQ[i]+sQ[3+j+i]*(long)hI[i];
   }
   yI[j/3]=(SumI+0x4000)>>14;          // with gain 2
   yQ[j/3]=(SumQ+0x4000)>>14;          // with gain 2

   // generate error
   if (yI[j/3]>=0){v.I=0x0800;}
   else {v.I=-0x0800;}
   if (yQ[j/3]>=0){v.Q=0x0800;}
   else {v.Q=-0x0800;}

   e.I=(v.I-yI[j/3])>>4;
   e.Q=(v.Q-yQ[j/3])>>4;

   // adaptation
   for (i=0; i<h_Leng; i++)
   {  SumI=e.I*(long)sI[3+j+i]+e.Q*(long)sQ[3+j+i];
      SumQ=e.Q*(long)sI[3+j+i]-e.I*(long)sQ[3+j+i];
      SumI=((SumI+0x4000)>>15)+hI[i];
      SumQ=((SumQ+0x4000)>>15)+hQ[i];

      hI[i]=(short)SumI;
      hQ[i]=(short)SumQ;
```

```
      // check for underflow and overflow and saturate
      if (SumI>0x7fff)    { hI[i]=0x7fff; }
      if (SumI<-0x7fff)   { hI[i]=-0x7fff; }
      if (SumQ>0x7fff)  { hQ[i]=0x7fff; }
      if (SumQ<-0x7fff)   { hQ[i]=-0x7fff; }
   }
}
```

Example 4: Fixed point version of a 1/3 T Software Equalizer

```
for (j=0; j<x_Leng; j=j+3)
{  for (SumI=0, SumQ=0, i=0; i<h_Leng; i++)
   {
      SumI = SumI+sI_float[3+j+i]*hI_float[i]-sQ_float[3+j+i]*hQ_float[i];
      SumQ = SumQ+sI_float[3+j+i]*hQ_float[i]+sQ_float[3+j+i]*hI_float[i];
   }
   yI_float[j/3]=(float)(2*(SumI));            // with gain 2
   yQ_float[j/3]=(float)(2*(SumQ));            // with gain 2

   // generate error
   if (yI_float[j/3]>=0){vI=0x0800/IFscale;}
   else {vI=-0x0800/IFscale;}
   if (yQ_float[j/3]>=0){vQ=0x0800/IFscale;}
   else {vQ=-0x0800/IFscale;}

   eI = (vI-yI_float[j/3])/(16.0f);            // step size
   eQ = (vQ-yQ_float[j/3])/(16.0f);            // step size

   // adaptation
   for (i=0; i<h_Leng; i++)
   {
      hI_float[i] = (float)(hI_float[i] + eI*(double)sI_float[3+j+i]
                  + eQ*(double)sQ_float[3+j+i]);
      hQ_float[i] = (float)(hQ_float[i] + eQ*(double)sI_float[3+j+i]
                  - eI*(double)sQ_float[3+j+i]);
   }
}
```

Example 5: Floating point version of a 1/3 T Software Equalizer

6 Streaming SIMD Extensions Assembly Code Example

The following example shows the Streaming SIMD Extensions inline assembly version of the 1/3 T Software Equalizer.

```
__asm
```

```
{
  mov   ecx, 0             ; intialize the outer loop counter
  mov   eax, hQI_float
  mov   edi, yIQ_float

  X_LOOP:
    cmp   ecx, last_x      ; check loop counter and jump conditionally
    jge   X_DONE

    mov   esi, ecx
    imul  esi, 3
    add   esi, sIIQQ_float

    mov   edx, 0              ; initialize the inner loop counter
    xorps xmm7, xmm7          ; zero out the accumulator xmm_register

    H_LOOP:
      cmp   edx, last_h       ; check loop counter and jump conditionally
      jge   H_DONE

      movaps  xmm0, [esi][edx*2][48]   ; xmm0 <- | sI1 | sI1 | sQ1 |-sQ1 |
      movaps  xmm1, [esi][edx*2][64]   ; xmm1 <- | sI2 | sI2 | sQ2 |-sQ2 |
      movaps  xmm2, [eax][edx]         ; xmm2 <- | hI2 | hQ2 | hI1 | hQ1 |
      movaps  xmm3, xmm2               ; xmm3 <- | hI2 | hQ2 | hI1 | hQ1 |

      shufps  xmm2, xmm2, 0x44         ; xmm2 <- | hI1 | hQ1 | hI1 | hQ1 |
      shufps  xmm3, xmm3, 0xEE         ; xmm3 <- | hI2 | hQ2 | hI2 | hQ2 |

      mulps   xmm0, xmm2 ; xmm0 <- | sI1*hI1 | sI1*hQ1 | sQ1*hI1 |-sQ1*hQ1 |
      mulps   xmm1, xmm3 ; xmm1 <- | sI2*hI2 | sI2*hQ2 | sQ2*hI2 |-sQ2*hQ2 |

      addps   xmm7, xmm0
      addps   xmm7, xmm1

      movaps  xmm3, [esi][edx*2][80]   ; xmm3 <- | sI3 | sI3 | sQ3 |-sQ3 |
      movaps  xmm4, [esi][edx*2][96]   ; xmm4 <- | sI3 | sI3 | sQ3 |-sQ3 |
      movaps  xmm5, [eax][edx][16]     ; xmm5 <- | hI4 | hQ4 | hI3 | hQ3 |
      movaps  xmm6, xmm5               ; xmm6 <- | hI4 | hQ4 | hI3 | hQ3 |

      shufps  xmm5, xmm5, 0x44         ; xmm5 <- | hI3 | hQ3 | hI3 | hQ3 |
      shufps  xmm6, xmm6, 0xEE         ; xmm6 <- | hI4 | hQ4 | hI4 | hQ4 |

      mulps   xmm3, xmm5 ; xmm2 <- | sI3*hI3 | sI3*hQ3 | sQ3*hI3 |-sQ3*hQ3 |
```

```
  mulps   xmm4, xmm6 ; xmm4 <- | sI4*hI4 | sI4*hQ4 | sQ4*hI4 |-sQ4*hQ4 |

  addps   xmm7, xmm3
  addps   xmm7, xmm4          ; xmm7 <- | sI*hI | sI*hQ | sQ*hI |-sQ*hQ |

  add     edx, 32
  jmp     H_LOOP
H_DONE:

movaps  xmm0, xmm7            ; xmm0 <- | sI*hI | sI*hQ | sQ*hI |-sQ*hQ |
shufps  xmm0, xmm7, 0x1B      ; xmm0 <- |-sQ*hQ | sQ*hI | sI*hQ | sI*hI |
addps   xmm0, xmm7            ; xmm0 <- | SumI | SumQ | SumQ | SumI |
addps   xmm0, xmm0            ; xmm0 <- | yI | yQ | yQ | yI |
movaps  [edi][ecx], xmm0      ; yIQ[j/3] <- | yI | yQ | yQ | yI |

xorps   xmm2, xmm2            ; zero out an xmm_register for
                              ; comparison to zero
movaps  xmm1, xmm0            ; copy the output result (yIQ) from
                              ; above calculations
cmpps   xmm0, xmm2, 1         ; generate a "less than" mask
movaps  xmm5, xmm0            ; copy mask
movaps  xmm3, neg_const.vec   ; load two alternative outcomes of
                              ; if..else statement
movaps  xmm4, pos_const.vec

andps   xmm0, xmm3            ; vI and vQ calculated without
                              ; conditional branching
andnps  xmm5, xmm4
orps    xmm0, xmm5            ; xmm0 <- | vI | vQ | vQ | vI |

subps   xmm0, xmm1            ; xmm0 <- | vI-yI | vQ-yQ | vQ-yQ | vI-yI |
movaps  xmm5, step_size.vec   ; xmm5 <- | step | step | step | step |
mulps   xmm0, xmm5            ; xmm0 <- | eI | eQ | eQ | eI |

mov     edx, 0

  I_LOOP:
    cmp     edx, last_h      ; check loop counter and jump conditionally
    jg    I_DONE

    movaps  xmm2, [esi][edx*2][48]  ; xmm2 <- | sI1 | sI1 | sQ1 | -sQ1 |
    movaps  xmm3, [esi][edx*2][64]  ; xmm3 <- | sI2 | sI2 | sQ2 | -sQ2 |
    movaps  xmm4, [eax][edx]        ; xmm4 <- | hI2 | hQ2 | hI1 | hQ1 |
```

```
        mulps   xmm2, xmm0   ; xmm2 <- | eI*sI1 | eQ*sI1 | eQ*sQ1 |-eI*sQ1 |
        mulps   xmm3, xmm0   ; xmm3 <- | eI*sI2 | eQ*sI2 | eQ*sQ2 |-eI*sQ2 |

        movaps  xmm5, xmm2   ; xmm5 <- | eI*sI1 | eQ*sI1 | eQ*sQ1 |-eI*sQ1 |
        shufps  xmm5, xmm5, 0x4E      ; xmm5 <- | eQ*sQ1 |-eI*sQ1 | eI*sI1 |
                                      ;  eQ*sI1 |
        shufps  xmm2, xmm3, 0x44      ; xmm5 <- | eQ*sQ2 |-eI*sQ2 | eQ*sQ1 |
                                      ; -eI*sQ1 |
        shufps  xmm5, xmm3, 0xE4      ; xmm5 <- | eI*sI2 | eQ*sI2 | eI*sI1 |
                                      ;  eQ*sI1 |

        addps   xmm5, xmm2
        addps   xmm4, xmm5        ; xmm4 <- | hI2' | hQ2' | hI1' | hQ1' |

        movaps  [eax][edx], xmm4  ; hIQ  <- | hI2' | hQ2' | hI1' | hQ1' |

        add     edx, 16
        jmp     I_LOOP
      I_DONE:

      add    ecx, 16
      jmp    X_LOOP
    X_DONE:
}
```